Scientia Iranica (2024) 31(2), 83–102

Sharif University of Technology
Scientia Iranica
Transactions D: Computer Science & Engineering and Electrical Engineering
https://scientiairanica.sharif.edu

Doctor Code: A machine learning-based approach to program repair

Sh. Moosavi, M. Vahidi-Asl*, H. Haghighi, and M. Rezaalipour

Faculty of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran.

Received 26 October 2019; received in revised form 19 April 2023; accepted 8 August 2023

KEYWORDS
Program repair;
Patch generation;
Machine learning;
Patch overfitting;
Multinomial logistic regression.

Abstract. In the last decade, several Automated Program Repair (APR) techniques have been proposed. Most of these techniques produce candidate patches using repair operators that apply changes to buggy programs. Therefore, to repair more bugs, a more extensive set of repair operators is required. However, more repair operators lead to longer repair times and more overfitted patches. To address the above-mentioned issues, we present Doctor Code, a new APR technique that chooses repair operators by systematically learning from the features of the most common bugs in different programs based on machine learning. We compare our technique against Mutation repair by the Siemens suite. The experiment results indicate that our technique can fix 41 bugs while the baseline only repairs 22. In addition, Doctor Code can produce patches that do not exist in the search space of SPR, Prophet, and SemFix. We also tested Doctor Code utilizing three buggy versions of a program called Space to indicate its capability of repairing large-sized programs. Also, we compare Doctor Code against 7 state-of-the-art APR tools, like Elixir, using the Defects4j dataset. The experiment results indicate that our technique outperforms the other tools in terms of the number of fixed bugs and overfitted patches.

© 2024 Sharif University of Technology. All rights reserved.

1. Introduction

Program repairing is an important activity to fix bugs during software development. In recent years, various automated techniques have been proposed to locate software bugs [1]. However, applying these techniques requires human developers to fix the located bugs. Manual repair is an inaccurate, frustrating, and costly process. In addition, in some critical software applications, the absence of automatic program repair mechanisms may lead to irreparable physical and financial losses [2]. These challenges have strongly motivated researchers to seek Automated Program Repair (APR) techniques. Different kinds of APR techniques have been established in the literature, including patch generation [3–8] and runtime repair [9,10].

There is a class of APR techniques that utilize test suites to generate patches [4]. Test suite-based APR techniques produce a set of candidate patches and evaluate them with an existing test suite until a patch is found, which makes failing test cases pass while keeping the passing ones satisfied. To produce a set of candidate patches, APR techniques employ several repair operators, each of which performs syntactical modifications on the buggy program. Each repair operator can fix a few types of bugs. Therefore, to fix more bugs, the number of repair operators

*. *Corresponding author. Fax: +98 21 22431607*
E-mail address: mo_vahidi@sbu.ac.ir (M. Vahidi-Asl)

doi: 10.24200/sci.2023.54718.3884

must be increased. However, employing more repair operators results in a larger patch search space, which raises two different problems for a test suite-based APR technique. First, since its patch search space is extensive, it has to evaluate more candidate patches to find a patch that satisfies the given test suite, and thus, repairing each bug takes more time. Second, the likelihood of producing patches that overfit test suites (overfitted patches) rises with growth in the patch search space [11]. The overfitted patches are produced due to the incompleteness of test suites. Utilizing these patches results in programs that, despite being incorrect, satisfy every test case in the existing test suite [12].

To remedy the above-mentioned problems, the number of candidate patches needs to be declined. One possible solution is to find the right repair operator that is capable of producing the correct patch for the bug at hand. We hypothesize that there is information in the buggy program and the bug itself that can aid APR techniques in finding the right repair operator among a long list of ones they possess. Finding the right repair operator to repair a bug is a non-deterministic problem, and there is no specific heuristic to address that. Therefore, solving it using machine learning-based approaches may lead to promising results [8,13,14]. Using machine learning, APR techniques can produce patches for new bugs based on the models constructed at the training phase from similar bugs and their corresponding patches. The intuition is that similar bugs have similar patches. Thus, they can be repaired using similar repair operators.

In this paper, we present a novel machine learning-based APR technique called Doctor Code. The specific feature of Doctor Code, in comparison with the earlier techniques, which use repair operators in a fixed order and regardless of the bug types, is that it chooses repair operators by systematically learning from the features of the most common bugs in different programs. Utilizing a trained Multinomial Logistic Regression (MLR) model, Doctor Code prioritizes its repair operators according to the bug at hand. With this prioritization method, the candidate patches are more accurately selected, and bugs are repaired, evaluating fewer candidate patches. In this way, by producing fewer candidate patches, the speed of the repair process increases, and the likelihood of producing overfitted patches reduces [11], leading to increased repair rates. The main contributions of this paper are:

- Designing ten repair operators, each of which is capable of fixing a specific type of bug;
- Proposing a method that employs machine learning to prioritize repair operators.

We implement Doctor Code to repair programs using C and Java language and evaluate it using the benchmark programs inside the Siemens suite [15] and Defects4j [16]. To evaluate the performance of Doctor Code, we have conducted five experiments. In the first experiment, we compared Doctor Code with Mutation repair [6]. According to the results, our technique can produce 41 correct patches for the Siemens suite, while Mutation repair [6] fixes 22. The three techniques SPR [7], Prophet [8], and SemFix [5] repair programs in the C language as well. Found by analysis, Doctor Code is capable of producing patches that do not exist in the search space of these three techniques.

In addition, we have compared Doctor Code against 7 state-of-the-art APR tools using the Defects4j dataset. The experiment results indicate that Doctor Code outperforms the other tools in terms of the number of fixed bugs. In the third experiment, to indicate that Doctor Code is capable of repairing large-sized programs, we evaluated it on three buggy versions of the Space program (about 9K LOC) [15]. According to the results, Doctor Code can produce the correct patches for all three of them. The fourth experiment evaluates the second contribution, the prioritization capability of the proposed technique. To do so, we have implemented another APR technique called Random APR (RAPR) using the same repair operators Doctor Code employs. Comparing Doctor Code with RAPR as the baseline indicates that using machine learning to prioritize repair operators reduces the repair time and the number of overfitted patches by 82.68% and 33.33%, respectively. In the last experiment, to evaluate the effectiveness of MLR, we used Support Vector Machine (SVM), random forrest, and Artificial Neural Networks (ANN) models instead of MLR in the training phase. The results indicate that MLR outperforms other models in terms of running time.

The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 presents the proposed technique of this paper. Section 4 provides the evaluation of the proposed technique, and Section 5 concludes this work.

2. Related work

In this section, we review famous studies in APR. Arcuri [17] was the first to use genetic programming to design an APR technique. Marshall and Wallace [3] and Goues et al. [4] extended this approach and presented GenProg, a test suite-based APR technique that repairs buggy programs using genetic programming. However, as stated in [18], the main issue with genetic programming is the high amount of data to search through to identify the right program. This concern cannot be solved in GenProg and is always there. In addition, this approach applies only genetic operators

to high-granularity modifications, and additional edits are not generated by its crossover operator.

Several studies have been conducted to improve GenProg [3,13,19]. The main challenges of these methods include their low scalability and generality. By scalability, we mean how quickly methods find fixes and how many lines of code they can handle. The generality of a method is related to the variety of programs and bugs that the methods can address.

The idea behind employing evolutionary algorithms for bug fixing is quite novel and outstanding. However, according to the study conducted by Qi et al. [12], most of the patches produced by GenProg [3] and AE [19] are either incorrect or overfitted, and those few correct patches produced by these techniques were the result of removing statements from the buggy programs. On the other hand, it is worth mentioning that the empirical study of Qi et al. [12] was one of the initial steps toward the definition of the patch overfitting problem, and patch overfitting was not discussed before 2015, which is when these studies have been published.

Kim et al. [20] proposed Pattern-based APR (PAR). They discovered a list of fixed patterns and templates by reviewing 60,000 human-written patches. Utilizing these templates and employing genetic programming, PAR repairs Java programs. To the best of our knowledge, PAR [20] is the first technique that proposes the use of templates to repair bugs, which is a sound idea. However, based on a critical review presented by Monperrus [21], PAR repairs most of its bugs by only using a limited number of the templates it possesses, and thus, it is capable of fixing a few types of bugs.

Debroy and Wong [22] proposed Mutation repair, a test suite-based APR technique, which is the result of combining various mutation testing operators and a fault localization technique such as Tarantula [23] and Ochiai [24]. Mutation repair generates a set of ranked statements using a fault localization technique. Then, it employs eight mutation testing operators to produce candidate patches for each statement in this set. However, the mutation operators used by Mutation repair are very simple. As a result, it can only fix very simple bugs. Also, Mutation repair employs a limited number of mutation operators for bug fixing since it does not have a mechanism for choosing the appropriate one for the bug at hand. Thus, extending it to repair more bug types cannot be simply done by adding new mutation operators to the list of its repair operators, which is not the case for Doctor Code.

Long and Rinard [7] presented SPR, which repairs bugs in conditional statements by condition synthesizing. It also fixes other types of bugs, such as missing memory initialization statements. SPR generates a set of parametric candidate patches for a given buggy program using a list of parametric transformation schemas. Then, candidate patch parameters are synthesized using a test suite that has at least one failing test case. These schemas are the repair operators of SPR and are applied to the bug location in a pre-defined order. Later, Long and Rinard [8] extended SPR and presented Prophet. Prophet utilizes the repair operators of SPR and ranks candidate patches using a probabilistic model, trained with a set of buggy programs and their patches. Experiments indicate that by reducing the number of overfitted patches, Prophet produces more correct patches than SPR. Section 4.2.4 thoroughly reviews the limitations of these two techniques in comparison with Doctor Code.

Nguyen et al. [5] proposed SemFix, a constrained-based APR technique, to repair bugs existing at the right-hand side of assignment statements and those occurring in conditional statements of programs. SemFix utilizes symbolic execution techniques to derive constraints from existing test cases. Then, employing component-based program synthesis, SemFix solves these constraints to generate a concrete patch that replaces the buggy statement. Section 4.2.4 thoroughly reviews the limitations of SemFix in comparison with Doctor Code. Nopol [25], an APR technique inspired by SemFix, repairs bugs appearing in conditional statements of Java programs. This technique receives a buggy program and a test suite (containing at least one failing test case) as input and employs the angelic debugging technique and execution traces produced from running the program with the test suite to generate a patch for the bug at hand. The limitation of Nopol is that it can only fix bugs that are repairable by updating the conditional statement of an *if* block or adding a new if statement to the buggy program. Doctor Code can repair various other bug types as well as these two types of bugs.

Saha et al. [14] proposed Elixir by focusing on repairing method-invocation-related bugs in Object-oriented Java programs. Employing a list of program transformation schemas and utilizing a patch ranking mechanism, Elixir produces a set of ranked candidate patches. Then, candidate patches are evaluated one at a time against a test suite to find a patch that satisfies the whole test case. Although Elixir is effective at fixing bugs caused by incorrect method invocations in Object-oriented programs, there are several other bug types it cannot fix. For instance, Elixir is not capable of fixing bugs caused by incorrect use of logical or assignment operators. It also does not attempt to repair literal-related bugs. In addition, none of the techniques Elixir, SPR, Prophet, Nopol, and SemFix can fix bugs that require tightening or loosening of conditional statements using expressions as complex as

"$(!f1(arr1[*a], b, c))$", where $f1$ is a boolean function, $arr1$ is an integer array, and a, b, and c are integer variables. Doctor Code does not have these limitations.

DLFix is a tool for APR for fixing Java bugs [26]. This method is based on Deep Learning and uses a tree-based RNN (Recurrent Neural Network) with two layers. The first layer encodes the Abstract Syntax Tree (AST) that surrounds the buggy source code. Then, the encoded AST is passed to the second layer as a vector. This layer takes the context vector and learns how to transform the buggy sub-tree. It generates multiple patches for a single bug and deploys a character-level CNN (Convolutional Neural Network) to rank all the generated patches.

3. Doctor Code

This section presents Doctor Code, the proposed technique of this paper. Doctor Code is a test suite-based APR technique that utilizes machine learning techniques to reduce the number of candidate patches required to be evaluated before the first correct patch is found. By doing so, many candidate patches that cannot fix the bug at hand are not evaluated at all. Thus, the repair time and the number of overfitted patches decline.

3.1. Overview

The overall structure of Doctor Code is depicted in Figure 1, and it operates in two phases: training (Figure 1(a)) and deployment (Figure 1(b)). The training phase employs a list of buggy programs to train a prediction model capable of estimating the likelihood of bug repair employing each repair operator. The deployment phase patches a given buggy program using the prediction model.

As illustrated in Figure 1(a), the first phase comprises two stages, which are the feature extraction stage and the model construction stage. The goal of the model construction stage is to train the prediction model, which requires a training set. A training set is a list of instances, each of which is a pair of a feature vector and a label. To produce the feature vectors, we need a list of buggy programs whose bug locations are known. The goal of the feature extraction stage is to extract several features from each of these buggy programs automatically and produce the feature vectors. These buggy programs and their bug locations are two of the inputs to the training phase. The features extracted for each buggy program investigate the existence of different program components (e.g., assignment and relational operators) in the bug location, along with information regarding the characteristics of the statements surrounding the bug location, such as the return type of the buggy function.

To produce the labels of the instances in the training set, each bug in the given buggy programs is reviewed manually to realize which repair operator can fix it. The current version of Doctor Code has ten repair operators, each of which is assigned a number between 1 and 10 (see more details in Subsection 3.5.1).

As illustrated in Figure 1(b), the deployment phase comprises two stages named feature extraction and patch generation, and its goal is to propose a patch

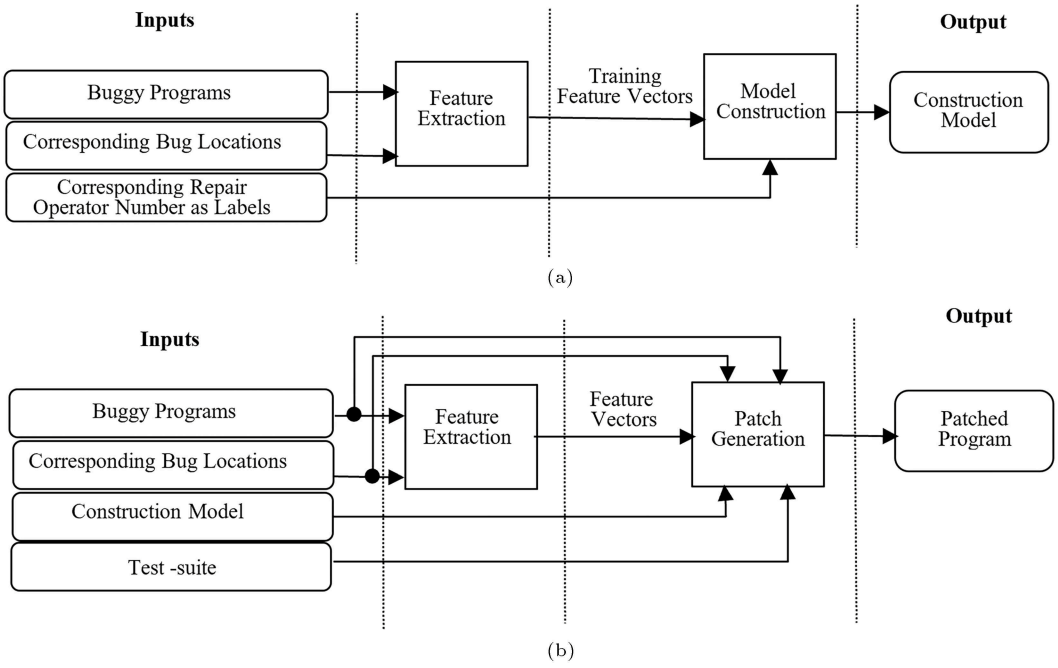

Figure 1. The overall structure of Doctor Code. (a) The training phase; (b) The deployment phase.

for a given buggy program. In this phase, first, a list of features is extracted from the buggy program. Then, using the resulting feature vector and the model constructed at the training phase, the repair operators of Doctor Code are prioritized and employed in order of their priority. Applying each repair operator results in a set of candidate patches, which are evaluated against the given test suite. The first candidate patch that satisfies the whole test suite is selected as the result of this phase.

In the rest of this section, we explain Doctor Code in detail. To this end, Section 3.2 provides a buggy program, Section 3.3 explains the feature extraction stage, Section 3.4 presents the details of the model construction stage, and Section 3.5 explains the patch generation stage.

3.2. Running example

Figure 2 illustrates a buggy code and its corresponding test suite, which we employ as a running example to explain the different stages of the proposed technique. Although this program is in the C language, the proposed technique is not specific to C and can repair programs in different procedural languages.

The program in Figure 2(a) comprises three functions with the names *main*, *min*, and *range*, and also a variable named d defined and initialized in the global scope. This program has a bug in line 29. The developer has written the condition of the *if* statement in line 29 as "$(c \leq b)$", while the correct form is "$(c \leq a$ && $c \leq b)$". In this paper, we refer to bug-containing functions as the buggy function. Figure 2(b) shows the test suite of this program, including two test cases.

3.3. Feature extraction

Both the training and deployment phases of Doctor Code have the feature extraction stage in which a list of features is extracted from buggy programs to produce their feature vectors. The training phase employs these feature vectors to train the prediction model, and the deployment phase uses them to estimate the likelihood of bugs being repaired by each repair operator.

One of the most challenging activities in proposing Doctor Code was to define an appropriate set of features to be extracted from the source code of a buggy program. To do so, we reviewed the features proposed by other similar techniques [8,13,14]. Then, we defined a list of features to be extracted from the source code of buggy programs. After performing some experiments using this list, we selected 18 features from

```
1:  #include <stdio.h>
2:  #include <stdlib.com>
3:
4:  int min( int a, int b, int c);
5:  int range(int a);\
6:
7:  int d = 12;
8:
9:  int main(int argc, char* argv[])
10: {
11:         int a = atoi( argv[1] );
12:         int b = atoi( argv[2] );
13:         int c = atoi( argv[3] );
14:         int mi = min(a, b, c);
15:         printf("%i", mi);
16:
17:         int r = range(a);
18:         printf("%i", r);
19:         return 0;
20: }
21:
22: int min (int a, int b, int c)
23: {
24:         int min = a;
25:         if ( b <= a && b <= c)
26:         {
27:                 min = b;
28:         }
29:         else if( c <= b)
30:         {
31:                 min = c;
32:         }
33:         return min;
34: }
35:
36: int range(int a)
37: {
38:         int r = a;
39:         int ar[2] = {10, 100};
40:         int i = 0;
41:         if(a < ar[i])
42:                 r= 10;
43:         else if(a > ar[i+1])
44:                 r = 100;
45:         return r;
46: }
```

(a)

#	Inputs a	b	c	Observed result	Expected result	Status
1	2	4	3	3.10	2.10	Fail
2	4	5	3	3,10	3.10	Pass

(b)

Figure 2. Example of a buggy program and its corresponding test suite. (a) Buggy program; (b) Test suite.

Table 1. List of features extracted from buggy programs.

ID	Description
	Buggy statements (10 features)
BS1	The buggy statement has numerical literals (e.g., 1, 2.1,...)
BS2	The buggy statement has character literals (e.g., '*a*')
BS3	The buggy statement is a function call
BS4	The buggy statement is a simple assignment (assigning a literal to a variable)
BS5	The buggy statement has a boolean expression
BS6	The buggy statement is a branch predicate (e.g., guard of an *if* statement)
BS7	The buggy statement has relation operators (e.g., >, <, ==, ...)
BS8	The buggy statement has logical operators (&&, ||)
BS9	The buggy statement has assignment operators (e.g., =, + =, − =, ...)
BS10	The buggy statement has an array of objects containing variables in their index expression
	Surrounding statements (8 features)
SS1	The buggy function returns an integer object
SS2	The buggy function returns a double object
SS3	The buggy function returns a character object
SS4	Buggy functions return a string object
SS5	The buggy function is void
SS6	The buggy function does not have arguments
SS7	The buggy statement is inside the global scope
SS8	The buggy statement is inside the main function

BS1	BS2	BS3	BS4	BS5	BS6	BS7	BS8	BS9
0	0	0	0	1	1	1	0	0
BS10	SS1	SS2	SS3	SS4	SS5	SS6	SS7	SS8
0	1	0	0	0	0	0	0	0

Figure 3. Feature vector produced for the bug in Figure 2(a), based on the features introduced in Table 1.

([1,0,0,0,1,1,1,0,0,0,0,0,0,0,1,0,0,1], 1)
([1,0,0,1,0,0,0,0,1,0,0,0,1,0,0,0,0,0],2)
([0,1,0,0,1,1,1,0,0,1,0,0,0,0,1,0,0,0],7)
([1,0,0,0,1,1,1,0,0,0,1,0,0,0,0,0,0,1],8)
...

Figure 4. Example of a training set.

it (illustrated in Table 1). Figure 3 shows an example of a feature vector that is produced for the program in Figure 2(a). Features of the first set are extracted from buggy statements, and features of the second set are extracted from statements surrounding buggy statements.

3.4. Model construction

The goal of the model construction stage is to train a classifier and, by doing so, construct a prediction model. Since we have multiple repair operators to choose from, a multi-class classifier is required, and to train a machine learning-based classifier, a training set is needed. Figure 4 illustrates an example of a training set. As can be seen in this figure, each row in the training set, also called an instance, is a tuple that comprises a feature vector as its first element and a number as its second element. The second element in an instance is the number of repair operators that can fix the buggy program from which the feature vector in the instance is extracted. These numbers are used as labels while training the prediction model.

We employ the MLR technique [27], which is a multiclass classifier technique, as our prediction model. Providing a set of independent variables, MLR predicts the probabilities of each possible outcome (class) of a categorically distributed dependent variable. For example, in the proposed technique, the 18 features in Table 1 are the independent variables, and the repair operator numbers are considered the categorically distributed dependent variable. If the proposed technique possesses K repair operators, the dependent

variable may have K different outcomes. Therefore, the trained MLR predicts K different probabilities for the feature vector of a buggy program. Doctor Code employs these probabilities to prioritize its repair operators. Since MLR is capable of producing such probabilities, we have chosen it as the prediction model of our technique.

In our experiments, we use the Weka toolkit [28] for the training process and model construction. By using the Java classes of Weka and a training set similar to the one in Figure 4, we train an MLR-based prediction model to be used for patch production.

3.5. Patch generation

The goal of the patch generation stage is to apply repair operators to a buggy program in an appropriate order until a patch is found that satisfies the given test suite. We reviewed different APR-related studies and also examined an extensive set of bugs along with the patches proposed for them, and then we designed a list of ten repair operators. In the following, we first present the repair operators of Doctor Code (see Subsection 3.5.1), and then, we describe the algorithm of the patch generation stage (see Subsection 3.5.2).

3.5.1. Repair operators

The patch generation stage employs the ten repair operators illustrated in Table 2 to apply various modifications to buggy programs. Relying on the plastic surgery hypothesis [29], Doctor Code employs program elements of the given buggy program to synthesize a rich set of expressions called the Expression Set (ES). Members of ES are used by those repair operators that require synthesized expressions to fix bugs (i.e., predicate patching repair operators). To produce ES, first, an operand set is formed for the bug at hand, which includes:

1. Variables in the scope of the bug location;
2. Literals 0, 1, and -1;
3. Numeric and character literals that exist in the buggy function and the global scope.

For example, $\{a, b, c, \min, d, 0, 1, -1, 12\}$ is the operand set produced for the bug in Figure 2(a). Afterward, using members of the operand set, ES is produced, which involves:

1. Six logical expressions synthesized for each pair of the operand set members, using the six relational operators in the set $\{f <, \leq, >, \geq, ==, ! = g\}$;
2. Operands of logical operators, existing in conditional predicates of control flow statements, within the buggy function, and also the negation of these operands;
3. Conditional predicates of control flow statements existing in the buggy function and the negation of these predicates.

For example, Eq. (1) shows the ES synthesized for the bug in Figure 2(a) based on the operand set mentioned above.

Table 2. Repair operators of Doctor Code.

ID	Description
	Literal patching
1	Replacing a literal 1 with 1, 0, −1, l+1, and l−1
2	Replacing a literal with every literal within the buggy function and the global scope
3	Incrementing/decrementing the index expression of arrays
	Predicate patching
4	Removing logical operands
5	Appending a logical expression to a buggy statement, along with &&
6	Appending a logical expression to a buggy statement, along with ||
7	Appending a guard precondition (i.e., *if* statement)
	Operator patching
8	Changing a relational operator with every member of the same class
9	Changing a logical operator with every member of the same class
10	Changing an assignment operator with every member of the same class

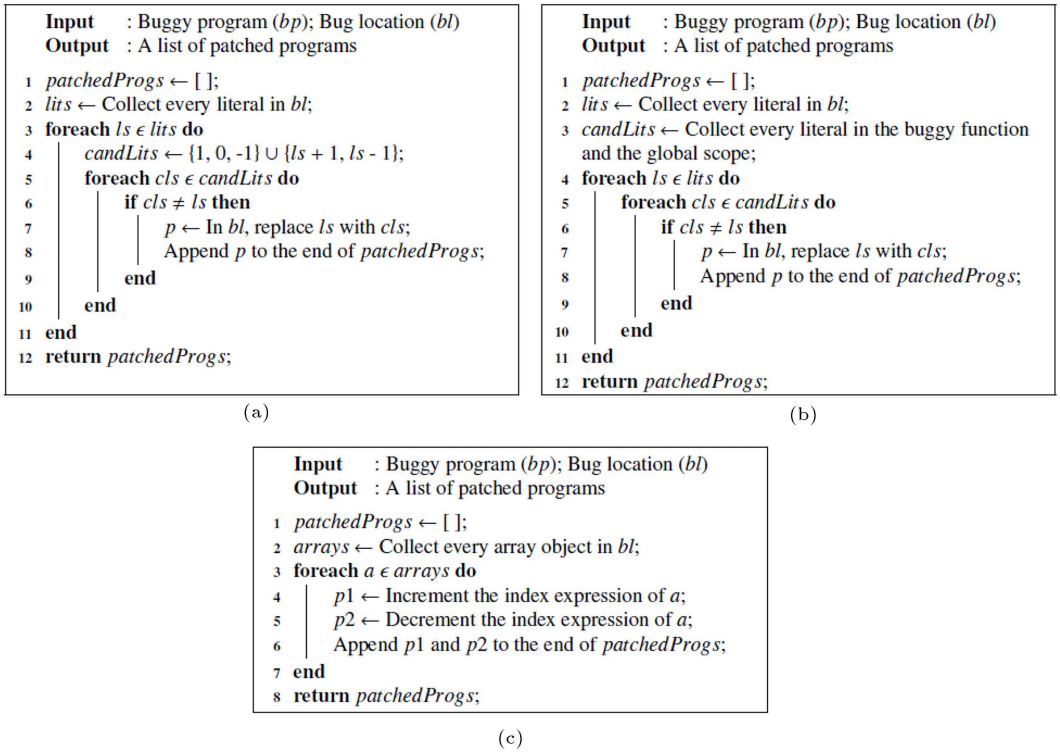

Figure 5. Literal patching repair operators. (a) Repair operator 1; (b) Repair operator 2; and (c) Repair operator 3.

$$
\begin{aligned}
ES = \{&a > b, a < b, a \geq b, a \leq b, a \neq b, a == b,\\
&a > c, a < c, a \geq c, a \leq c, a \neq c, a == c,\\
&\ldots,\\
&b > c, b < c, b \geq c, b \leq c, b \neq c, b == c,\\
&\ldots,\\
&c > \min, c < \min, c \geq \min, c \leq \min,\\
&c \neq \min, c == \min,\\
&\ldots,\\
&d > 0, d < 0, d \geq 0, d \leq 0, d \neq 0, d == 0,\\
&\ldots,\\
&b \leq a, b \leq c, !(b \leq a), !(b \leq c), (b \leq a \;\&\&\; b \leq c),\\
&!(b \leq a \;\&\&\; b \leq c)\}. \qquad (1)
\end{aligned}
$$

As illustrated in Table 2, we categorize the repair operators of Doctor Code into three groups of literal patching, predicate patching, and operator patching. These repair operators receive a buggy program along with its bug location as input and return a sequence of patched programs.

- **Literal patching (repair operators 1–3).** We have designed three repair operators that attempt to fix bugs by performing literal modifications. Programmers sometimes make mistakes in defining variable boundaries. For example, loop variables are often initialized with a value that is one unit greater (or less) than what it should be. Fixing these bugs is the goal of the first repair operator (Figure 5(a)). However, not every literal-related bug can be fixed using the literal candidate set produced by the first repair operator. For example, consider a buggy program that is only fixed when the expression "$(a > 50)$" is changed into "$(a > 500)$". The second repair operator attempts to fix these bugs (Figure 5(b)). The third repair operator increments or decrements the index expression of each array in the bug location, and with every change, it generates a new patched program (Figure 5(c)).

- **Predicate patching (repair operators 4–7).** Having examined several existing bugs and their patches, we realized that predicates are very prone to bugs. Empirical analysis studies on program repair also indicate that the majority of bugs are related to if statements [30]. Therefore, we designed four repair operators (repair operators 4–7) to repair predicates and if-related bugs. The fourth repair operator, depicted in Figure 6(a), generates patches by removing logical operands in the bug location, one at a time.

The fifth and sixth repair operators employ ES to fix bugs occurring in predicates (Figure 6(b) and

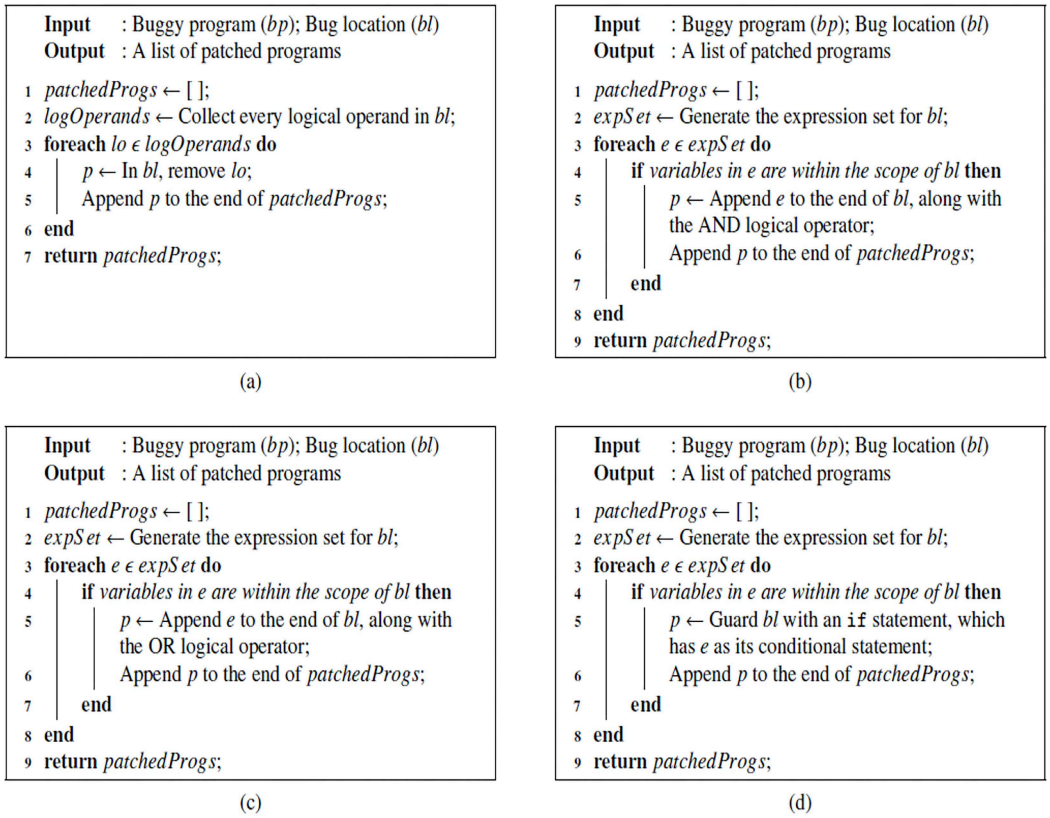

Figure 6. Predicate patching repair operators. (a) Repair operator 4; (b) Repair operator 5; (c) Repair operator 6; and (d) Repair operator 7.

6(c)). Finally, the seventh repair operator, shown in Figure 6(d), produces several patched programs by adding a guard precondition to the program.

- **Operator patching (repair operators 8–10).** We have three repair operators regarding the correction of bugs induced by selecting a wrong programming language operator (e.g., relational operators). Figure 7(a), (b), and (c) show the pseudo-codes of the eighth, ninth, and tenth repair operators, respectively.

By analyzing these ten repair operators, it might seem that some of these repair operators are overlapping in nature and could be merged. For example, both the first and the second repair operators try to fix bugs by replacing a literal with a different one. Although it is possible to merge these overlapping repair operators, we decided not to do so. The first reason behind this decision is that the whole idea behind Doctor Code is to reduce patch spaces wisely to decrease the possibility of overfitted patches being produced. Thus, having finer-grained repair operators is more desirable as they provide smaller design spaces. The second reason is that Doctor Code possesses a mechanism for prioritizing repair operators, and thus, it can employ a long list of repair operators and manage them using this prioritizing mechanism. Also, the patch spaces provided by these pairs of similar repair operators rarely contain similar patches, and thus, they target different bug types.

3.5.2. Algorithm description

Figure 8 presents the pseudo-code of the patch generation stage. Figure 9(a) shows an example of *ropProbs*, which indicates that repair operator five is the best choice for bug repair as it has the highest probability. Figure 9(b) illustrates *ropSeq* obtained by sorting *ropProbs* in Figure 9(a).

4. Evaluation

In this section, we evaluate Doctor Code. Section 4.1 explains the experiment setup. Section 4.2 provides the results of the experiments and analytically compares Doctor Code with SemFix [5], SPR [7], Prophet [8], and automatic machine learning methods.

4.1. Experiment setup

We selected benchmark programs in C++ and Java languages for our experiments, so two versions of Doctor Code were implemented, one for C++ and one for Java. In addition, we have implemented Doctor Code according to Figure 1 and conducted experiments on a virtual machine with two cores of Intel Core i5 CPU at 1.6 GHz, 2 GB memory, and a 64-bit version

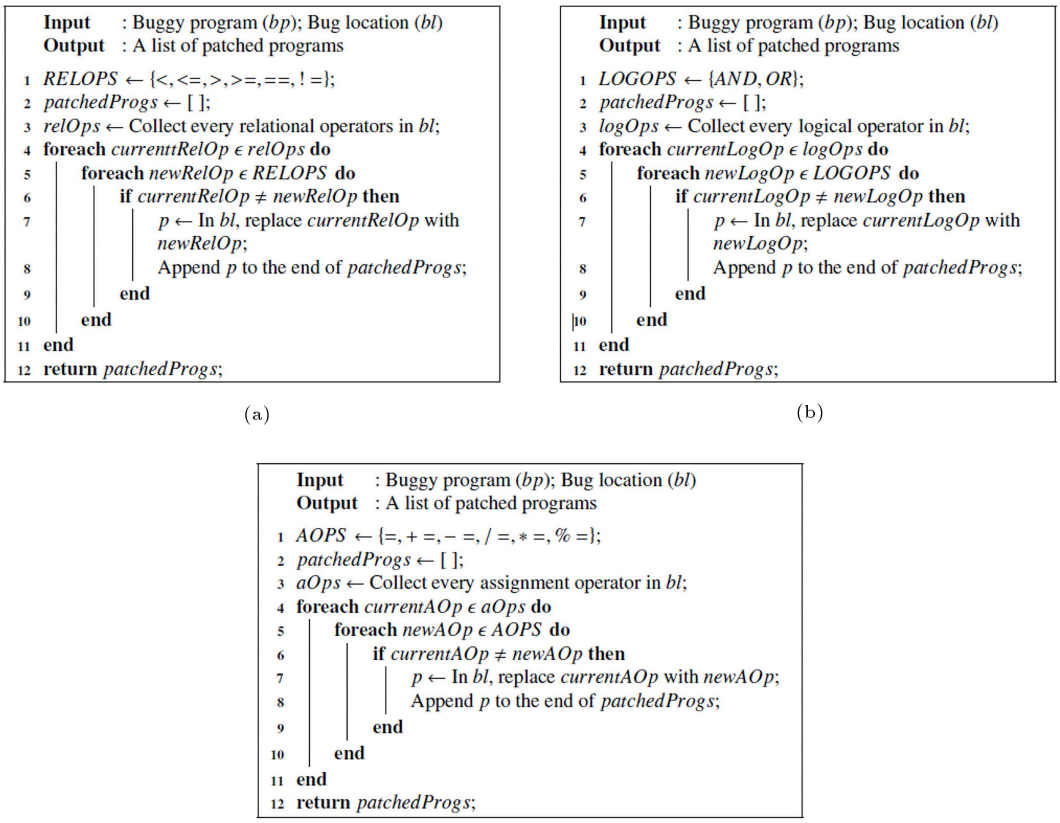

Figure 7. Operator patching repair operators. (a) Repair operator 8; (b) Repair operator 9; and (c) Repair operator 10.

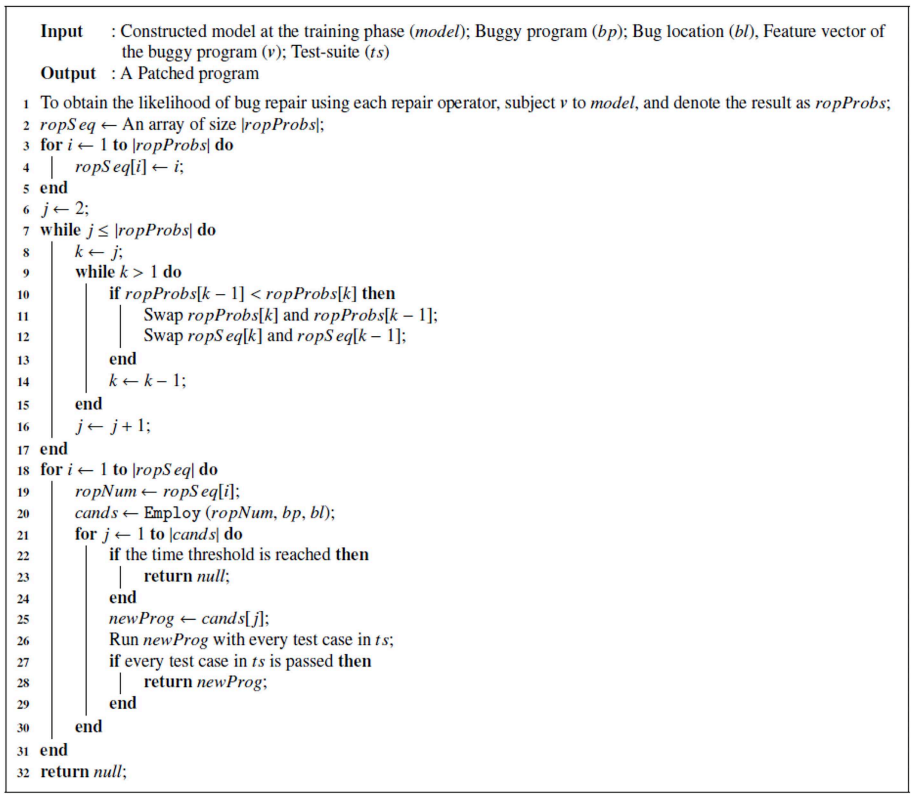

Figure 8. Patch generation stage.

Array Index (i.e., Repair Operator Number)	1	2	3	4	5	6	7	8	9	10
Likelihood	0.1	0.04	0.01	0.001	0.4	0.2	0.119	0.002	0.101	0.027

(a)

Repair Operator Sequence	5	6	7	9	1	2	10	3	8	4

(b)

Figure 9. Example of likelihoods estimated for the repair operators and the sequence of repair operator numbers obtained by sorting the computed likelihoods. (a) Likelihoods (ropProbs); (b) Repair operator sequence (ropSeq).

Table 3. Buggy programs from the Siemens suite.

Program	LOC	No. of buggy versions
Replace	564	18
Schedule	412	3
Schedule2	374	1
Tcas	173	12
Totinfo	565	10
Pronttokens2	570	6
Space	9564	3

Table 4. Buggy programs from Defects4J.

Program	LOC	No. of buggy versions
Commons math	85 k	106
Commons lang	22 k	65
Joda-Time	28 k	27
JFreeChart	96 k	26

of Ubuntu 14.04 LTS. We set the time threshold of the patch generation stage to 10 hours.

4.1.1. Subject buggy programs

To rigorously evaluate *Doctor Code*, we applied two sets of benchmark programs:

1. *Siemens suite:* We selected the Siemens suite [15] for our experiments because it has most of the bug patterns observed in our examination, and it is frequently used in other APR-related studies [5,6,30] and software testing papers [31,32]. Most of the patches made by earlier test suite-based APR techniques repair bugs that require single-point modifications. Therefore, in this paper, we do not intend to repair bugs that are fixed by modifying more than one point in programs. Thus, among the 132 buggy versions of the Siemens suite, we have selected 53 ones that have this characteristic. Table 3 illustrates the selected buggy programs that were used in our experiments;
2. *Defects4J:* To compare our method with existing automatic repair methods for Java programs, we selected Defects4J. Defects4J is a popular dataset that includes 395 Java buggy programs [16] against which existing machine-learned repair tools such as Elixir have been evaluated. Similar to Elixir, Doctor Code is evaluated using four subjects from Defects4J, and all the bugs that require multi-hunk fixes are discarded. Therefore, only 82 bugs that required a single-hunk fix were selected. Table 4 displays the details of this dataset.

As outlined in Section 3.3, Doctor Code extracts features only from buggy functions and global scopes of programs. Also, regarding Section 3.5.1, the repair operators of Doctor Code produce patches only using the elements inside buggy functions and global scopes. As a result, the size of buggy programs does not have a significant impact on Doctor Code. In fact, our technique is only affected by the size of the buggy function and the global scope of the given source code. However, to verify that our technique is capable of fixing large-sized programs, we also tested it using three buggy versions of Space [15] from the Siemens suite as well as JFreeChart from Defects4J.

4.1.2. Model construction

We followed the standard three-fold cross-validation methodology to train our prediction model. To this end, the 135 buggy versions, shown in Tables 3 and 4, were first divided into three mutually exclusive groups of buggy programs. Then, in three different trials, two groups were used to train the model, while the remaining one was employed for the evaluation.

4.1.3. Patch correctness

In this paper, a patch is considered correct if (1) the patched program satisfies the whole test suite and (2) applying it results in a code that is semantically equivalent to the correct version of the corresponding buggy program.

4.2. Experimental results

In this section, we report the results of applying Doctor Code on the subject buggy programs, presented in Section 4.1, to address the following questions:

- RQ1: How many correct and overfitted patches have been produced by Doctor Code?
- RQ2: What is the contribution of machine learning to reducing both the number of overfitted patches and repair time?

- RQ3: Is the MLR model the best prediction model to prioritize repair operators?
- RQ4: How effective is Doctor Code compared to automatic repair tools?

4.2.1. Number of patches (RQ1)

According to Tables 5 and 6, Doctor Code produced 41 correct patches for the Siemens suite, while Mutation repair [6] only produced 22. Doctor Code also successfully repaired all three buggy versions of Space, which reveals its capability to repair large-sized programs. These 44 correct patches were each produced in an average of 76.52 seconds. In addition, Doctor Code produced 36 correct patches for Defects4J, while Elixir, ACS, NOPOL, and jGenProg only produced 26, 18, 5, and 5 correct patches, respectively [26].

According to the third column of the tables, Doctor Code produced overfitted patches for 8 and 11 buggy versions of Siemens and Defects4j, receptively. After examining these overfitted patches and analyzing the repair operators, we realized that the patch search space of Doctor Code contains correct patches for these bugs, too. However, Doctor Code stopped before finding them since it found an overfitted patch for each of them, and due to insufficient coverage of the test suites, considered them as correct patches.

Considering the results, the proposed technique did not fix one of the Replace buggy versions within the specified time limit. However, after examination, it turned out that there is a correct patch for this bug in the search space of the repair operators. Therefore, if the test suite has enough coverage and Doctor Code is given enough time, it can repair this bug, too.

Table 7 illustrates the correct patches produced by Doctor Code for the Siemens suite. According to the results, most Doctor Code patches are identical to their corresponding correct versions.

4.2.2. Contribution of machine learning (RQ2)

The effectiveness of multi-class classification algorithms (e.g., MLR) is often evaluated using one of the two methods of micro-averaging or macro-averaging, including criteria such as Recall, F-score, and Precision [33]. However, these criteria are not suitable for assessing the effectiveness of the prediction model utilized by our technique. We elaborate on this issue by providing an example.

Figure 10 shows three repair operator sequences that the prediction model may generate at the patch generation stage while repairing the bug in Figure 2(a). The numbers in these sequences represent the repair operators in Table 2. Because this bug can be fixed using repair operator 5, and this operator has the highest priority in the first sequence in Figure 10, the prediction model is considered sufficiently effective if it generates the first sequence while repairing this bug. In the second sequence of Figure 10, repair

#	Repair Operator Sequence
1	5, 3, 2, 1, 4, 6, 10, 8, 9, 7
2	7, 5, 1, 4, 2, 8, 6, 9, 3, 10
3	1, 2, 5, 4, 9, 6, 3, 7, 10, 8

Figure 10. Examples of repair operator sequences produced at the patch generation stage.

Table 5. Information on patches produced by Doctor Code from Siemens.

Program	#Correct patches	#Overfitted patches	#Timeout
Replace	14	3	1
Schedule	2	1	0
Schedule2	1	0	0
Tcas	12	0	0
Totinfo	7	3	0
Pronttokens2	5	1	0
Space	3	0	0
Total	**44**	**8**	**1**

Table 6. Information on patches produced by Doctor Code from Defects4j.

Program	#Correct patches	#Overfitted patches	#Timeout
Math	16	4	0
Lang	10	3	1
Time	2	2	1
Chart	7	1	0
Total	**35**	**11**	**2**

Table 7. Correct patches produced by Doctor Code.

	Name	Buggy code	Correct code	Doctor Code patch
1	Printtokens2 V4	if ($ch == 59$) $id = 0$	if ($ch == 59$) $id = 2$	if ($ch == 59$) $id = 2$
2	Printtokens2 V5	return (True)	return (FALSE);	return (0)
3	Printtokens2 V7	if ($ch ==$ '\n' || $ch ==$ ' ')	if ($ch ==$ '\n')	if ($ch ==$ '\n')
4	Printtokens2 V8	if ($ch ==$ ' ' || $ch ==$ '\n' || $ch== 59$ || $ch ==$'\t')	if ($ch ==$ ' ' || $ch ==$ '\n' || $ch == 59$)	if ($ch ==$ ' ' || $ch ==$ '\n' || $ch == 59$)
5	Printtokens2 V9	if ($ch ==$ '\n' || $ch ==$'\t')	if ($ch ==$ '\n')	if ($ch ==$ '\n')
6	Replace V1	if (src[*i] == ESCAPE)	if (src[*$i-1$] == ESCAPE)	if (src[*$i-1$] == ESCAPE)
7	Replace V3	if ($m >= 0$)	if ($m >= 0$) && (lastm != m))	if ($m >= 0$ && (lastm != m)
8	Replace V9	else if (isalnum(src[*$i-1$])) && (isalnum(src[*$i+1$])))	else if (isalnum(src[*$i-1$])) && (isalnum(src[*$i+1$])) && (src[*$i-1$] <= src[*$i+1$]))	else if (isalnum(src[*$i-1$])) && (isalnum(src[*$i+1$])) && (src[*$i+1$] >= src[*$i-1$]))
9	Replace V13	if ($m == -1$) $i = i+1$; else $i = i+2$;	$i = i+1$	if ($m == -1$) $i = i+1$; else $i = i+1$;
10	Replace V14	if (lin[*i] != NEWLINE)	if (lin[*i] != NEWLINE) && (!locate(lin[*i], pat, j+1))	if ((lin[*i] != 10) && (!locate(lin[*i], pat, j+1)))
11	Replace V15	result = $i+1$;	result = i;	result = $i+0$;
12	Replace V16	return ($c ==$ BOL || $c ==$ EOL || $c ==$ CLOSURE || $c ==$ ANY);	return ($c ==$ BOL || $c ==$ EOL || $c ==$ CLOSURE);	return ($c ==$ BOL || $c ==$ EOL || $c ==$ CLOSURE);
13	Replace V17	result = NEWLINE;	result = ESCAPE;	result = 64;
14	Replace V18	if (!locate(lin[*i], pat, j+1)	if ((lin[*i] !=NEWLINE) && (!locate(lin[*i], pat, j+1)))	if ((lin[*i] != 10) && (!locate(lin[*i], pat, j+1)))
15	Replace V20	result = ENDSTR;	result = ESCAPE;	result = 64;
16	Replace V23	if (s[*i] == ENDSTR)	if (s[*$i+1$] == ENDSTR)	if (s[*$i+1$] == ENDSTR)
17	Replace V28	return ($c ==$ BOL || $c ==$ EOL || $c ==$ CLOSURE || $c ==$ COL);	return ($c ==$ BOL || $c ==$ EOL || $c ==$ CLOSURE);	return ($c ==$ BOL || $c ==$ EOL || $c ==$ CLOSURE);

Table 7. Correct patches produced by Doctor Code (continued).

Name	Buggy code	Correct code	Doctor code patch
18 Replace V29	return (c == BOL || c == EOL || c == CLOSURE || c == NCCL);	return (c == BOL || c == EOL || c == CLOSURE);	return (c == BOL || c == EOL || c == CLOSURE);
19 Replace V31	if ((lin[*i] >= NEWLINE) && (!locate (lin[*i], pat, j+1))	if ((lin[*i] != NEWLINE) && (!locate (lin[*i], pat, j+1)))	if ((lin[*i] ! = 10) && (!locate (lin[*i], pat, j+1)))
20 Schedule2 V7	if (ratio < 0.0 || ratio >= 1.0)	if (ratio < 0.0 || ratio > 1.0)	if (ratio < 0.0 || ratio >= 1.0)
21 Schedule2 V3	n = (int)(count*ratio+1, 1);	n = (int)(count*ratio+1);	n = (int)(count*ratio+1);
22 Schedule2 V9	if (argc < (MAXPRIO))	if (argc < (MAXPRIO+1))	if (argc < (3) || ((3) == argc))
23 Space V20	can = angle_step;	can + = angle_step;	can + = angle_step;
24 Space V21	cph = phase_step;	cph + = phase_step;	cph + = phase_step;
25 Space V33	gnode_ptr−>PHEA = angle;	gnode_ptr−>PHEA + = angle;	gnode_ptr−>PHEA + = angle;
26 Tcas V1	result = !(Own_Below_Threat()) || ((Own_Below_Threat()) && (!(Down_Separation > ALIM())))	result = !(Own_Below_Threat()) || ((Own_Below_Threat()) && (!(Down_Separation >= ALIM())))	result = !(Own_Below_Threat()) || ((Own_Below_Threat()) && (!(Down_Separation >= ALIM())))
27 Tcas V3	intent_not_known = TwoOfThreeReportsValid || Other_RAC == NO_INTENT;	intent_not_known = TwoOfThreeReportsValid && Other_RAC == NO_INTENT;	intent_not_known = TwoOfThreeReportsValid && Other_RAC == NO_INTENT;
28 Tcas V4	result = Own_Above_Threat() && (Cur_Vertical_Sep >= MINSEP) || (Up_ Separation>=ALIM());	result = Own_Above_Threat() && (Cur_Vertical_Sep >= MINSEP) && (Up_ Separation>=ALIM());	result = Own_Above_Threat() && (Cur_Vertical_Sep >= MINSEP) && (Up_ Separation>=ALIM());
29 Tcas V6	return (Own_Tracked_Alt <= Other_Tracked_Alt);	return (Own_Tracked_Alt < Other_Tracked_Alt);	return (Own_Tracked_Alt < Other_Tracked_Alt);
30 Tcas V9	Upward_preferred = Inhibit_Biased_Climb() >= Down_Separation;	Upward_preferred = Inhibit_Biased_Climb() > Down_Separation;	Upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
31 Tcas V12	enabled = High_Confidence || (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);	enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);	enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);

Table 7. Correct patches produced by Doctor Code (continued).

	Name	Buggy code	Correct code	Doctor code patch
32	Tcas V16	Positive_RA_Alt_Thresh[0] = 400+1;	Positive_RA_Alt_Thresh[0] = 400;	Positive_RA_Alt_Thresh[0] = 399+1;
33	Tcas V17	Positive_RA_Alt_Thresh[1] = 500+1;	Positive_RA_Alt_Thresh[1] = 500;	Positive_RA_Alt_Thresh[1] = 499+1;
34	Tcas V20	Upward_preferred = Inhibit_Biased_Climb() >= Down_Separation;	Upward_preferred = Inhibit_Biased_Climb() > Down_Separation;	Upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
35	Tcas V25	result = !(Own_Above_Threat()) || ((Own_Above_Threat()) && (Up_Separation > ALIM()));	result = !(Own_Above_Threat()) || ((Own_Above_Threat()) && (Up_Separation >= ALIM()));	result = !(Own_Above_Threat()) || ((Own_Above_Threat()) && (Up_Separation >= ALIM()));
36	Tcas V36	alt_sep = 1;	alt_sep = 2;	alt_sep = 2;
37	Tcas V38	int Positive_RA_Alt_Thresh[3];	int Positive_RA_Alt_Thresh[4];	int Positive_RA_Alt_Thresh[4];
38	Totinfo V5	totinfo = info;	totinfo += info;	totinfo += info;
39	Totinfo V9	totdf = infodf;	totdf += infodf;	totdf += infodf;
40	Totinfo V11	sum = del *= x / ++ap;	sum += del *= x / ++ap;	sum += del *= x / ++ap;
41	Totinfo V14	if (r *c >= MAXTBL)	if (r *c > MAXTBL)	if (r *c > 1000)
42	Totinfo V16	if (info >= 0.1)	if (info >= 0.0)	if (info >= 0.1 || ('\x00' <= info))
43	Totinfo V20	if (rdf <= 0)	if (rdf <= 0 || (cdf <= 0))	if (rdf <= 0 || (0>=cdf))
44	Totinfo V23	for (n = 0; n <=ITMAX; ++n)	for (n = 1; n <=ITMAX; ++n)	for (n = 1; n <=ITMAX; ++n)

operator 5 is prioritized as the second. So, if the second sequence is generated, the patches of repair operator 7 are produced, and these patches all fail, and then repair operator 5 is used. If this is the case, the prediction model is considered less effective compared to the scenario where the first sequence of Figure 10 is used. However, the second sequence is better than the third sequence, where repair operator 5 is prioritized as the third. Conventional criteria for evaluating multi-class classifiers do not distinguish between the second and the third sequences of Figure 10, and both are considered inappropriate (true negative). Therefore, they are not suitable for evaluating the prediction model of Doctor Code.

Therefore, to evaluate the prediction model, we examined the sequences of repair operators produced in the experiments. According to this examination, in 82.32% of the sequences, the correct repair operator was prioritized among the first three; in 74.62% of them, the correct one was prioritized among the first two; in 62.25% of the sequences, the correct repair operator was prioritized as the first item of the sequences. To further investigate the contribution of machine learning in prioritizing repair operators, we also implemented another APR technique called RAPR, in which repair operator sequences are produced randomly without employing the prediction model. Table 6 shows the results of running RAPR on the buggy programs of the Siemens suite introduced in Section 4.1. According to Tables 5 and 8, Doctor Code produced four fewer overfitted patches in comparison with RAPR. Additionally, RAPR timed out for two

Table 8. Information on patches produced by RAPR.

Program	#Correct patches	#Overfitted patches	#Timeout
Replace	13	4	1
Schedule	2	1	0
Schedule2	0	0	1
Tcas	10	2	0
Totinfo	7	3	0
Pronttokens2	5	1	0
Space	2	1	0
Total	**39**	**12**	**2**

Table 9. The percentage of the sequences where the correct repair operator is in the top three priority items, is in the top two priority items, and has the highest priority.

Prediction models	In the top three priority items	In the top two priority items	Has the highest priority
MultilayerPerceptron	81,13%	73.58%	58.49%
Random Forest	79.2%	71.69%	62.26%
MLR	81,13%	75.47%	62.26%
SVM	79.2%	73.58%	60.37%

Table 10. The produced sequences by the prediction models to Replace-v13.

Multilayer perceptron	Random forest	MLR	SVM
1 10 2 5 9 6 3 8 7 4	1 10 5 2 8 6 7 4 9 3	1 10 9 5 2 4 6 3 7 8	1 10 5 2 8 6 9 4 7 3

Table 11. General comparison between models.

Criteria	Multilayer perceptron	Random forest	MLR	SVM
Number of failing patches	3.68	2.73	2.47	2.57
The time to repair (in seconds)	84.57	81.42	76.52	83.74

bugs, while Doctor Code stopped once. Furthermore, RAPR produced each correct patch in an average of 56.5 minutes, while Doctor Code produced them in 9.68 minutes on average. Considering the results mentioned above, we can conclude that using machine learning to prioritize repair operators declines the number of overfitted patches and the time required to produce correct patches by $(12 - 8)/12 = 33{:}33\%$ and $(56{:}5 - 9{:}68)/56{:}5 = 82{:}68\%$, respectively.

4.2.3. The effectiveness of prediction models (RQ3)

We performed another experiment on the Siemens suit to evaluate the effectiveness of MLR. In this experiment, we used SVM, Random Forrest, and ANN models instead of MLR in the training phase. Table 9 shows the percentage of the sequences where the correct repair operator is in the top three priority items, is in the top two priority items, and has the highest priority. According to this examination, the models act almost identically to prioritize the correct repair operator among the first three items of the sequences. For example, the sequences of repair operators produced by each model for version 13 of the Replace buggy program are shown in Table 10. According to Table 10, the models assign the same priority to the first and second items while different for the lower priorities.

To investigate the subject further, we ran Doctor Code three more times by the new sequences of repair operators to compare the mentioned models in terms of running time. Running time and the number of failing patches to produce a correct patch, on average, are illustrated in Table 11. Although MLR performed better than the other models in terms of running time, Doctor Code showed almost the same performance for all four prediction models.

4.2.4. Comparison of Doctor Code with automatic machine learning methods (RQ4)

In this section, Doctor Code is compared to automatic program repair tools for C and Java programs, respectively.

Comparison with APR repair tools for C programs

SemFix. SemFix [5] has been successful at fixing the

Table 12. The number of generated correct/incorrect patches by the tools.

Programs	HD-repair	ACS	Nopol	SimFix (implemented by Java)	jGenProg	ELIXIR	DLFix	Doctor code
Commons math	1/21	12/16	1/21	14/26	5/18	12/19	12/28	16/4
Commons lang	3/7	3/4	3/7	9/13	0/0	8/12	5/12	10/3
Joda-Time	0/1	1/1	0/1	1/1	0/2	2/3	1/2	2/2
JFreeChart	0/2	2/2	1/6	4/8	0/7	4/7	5/12	7/1
Total	4/31	18/23	5/35	27/48	5/27	26/41	23/54	37/44
Percentage of overfitted patches to total patches	87%	21%	85%	77%	81%	36%	57%	18%

Table 13. Comparison between SemFix and Doctor Code in terms of time.

	Number of test inputs		Running time (in seconds)	
Programs	Doctor Code	SemFix	Doctor Code	SemFix
Tcas	1607	50	23	50
Schedule	2649	50	107	70
Replace	5541	50	126	70
Schedule2	2709	50	73	95

bugs in the Siemens suite. However, the correctness of the patches it produced has not been evaluated. Since the number of symbolic constraints of SemFix increases with a growth in the number of test cases (which slows down the repair process), it has experimented with a maximum of 50 test cases. On the other hand, the buggy versions of the Siemens suite have more than 1000 test cases. Limiting the number of test cases results in the production of overfitted patches [12]. Therefore, it is likely that some of the patches produced by SemFix might be overfitted patches. Doctor Code has no limitation on the number of test cases, and it is capable of evaluating candidate patches with thousands of test cases.

Moreover, the authors of [33] did a study to revisit the overfitting problem with a focus on semantics-based APR techniques, including SemFix. They performed the study on IntroClass and Codeflaws benchmarks. This study calculates the number of patches produced for each subject program that fail at least one held-out test for the IntroClass and Codeflaws datasets. On IntroClass and Codeflaws, 87% (86 of the 99) and 68% (38 of the 56) of patches generated by SemFix were overfitted to the training tests, respectively. The results are shown in Table 12, indicating that 43% of patches are overfitted. This suggests that, although semantics-based repair methods, including SemFix, have been shown to produce high-quality repairs on several subjects, overfitting to the training tests is still a concern for these approaches.

To perform a comparison between the tools in terms of running time, we re-implemented Doctor Code under the same conditions as the SemFix experiments (a Core 2 Quad 2.83 GHz CPU, 3 GB memory computer with Ubuntu 10.04 OS). It is not fair to compare these tools in terms of the running time because the number of test inputs for SemFix was at most 50, but the number of test inputs for Doctor Code was more than 50 (the number of test inputs is shown in Table 13). In the repair process, for each test input, the program under test is evaluated. In this way, for a large number of test inputs, the speed of the repair process decreases. The running time of the tools is shown in Table 13. Although the number of test inputs for Doctor Code was higher than SemFix, it outperformed SemFix for two cases (Tcas and Schedule2). Generally, despite a large number of test inputs for Doctor Code, its running time is similar to SemFix.

In addition, SemFix is only capable of repairing bugs that exist on the right-hand side of an assignment operator or bugs that occur in branch predicates. Doctor Code does not have this limitation. Furthermore, SemFix produces patches using symbolic execution. It is known that symbolic execution is problematic when the size and complexity of programs increase [25], which is not the case with Doctor Code. On the other

hand, since our technique relies on the plastic surgery hypothesis [29], it only repairs buggy programs whose patches can be produced using the existing program elements within corresponding buggy programs, which is not a limitation for SemFix.

SPR and Prophet. SPR [7] and Prophet [8] use six transformation schemas, three of which are to fix if-related bugs. These schemas are not capable of fixing bugs such as Replace V14 (row 10) in Table 5, where the conditional statement needs to be tightened with $!locate(lin[^*i], pat, j+1)$. Considering the same limitation, they cannot fix bugs such as Totinfo V20 (row 43), in which the conditional statement has to loosen utilizing ($cdf \geq 0$). Also, these schemas are designed to fix bugs that occur in the predicate of if statements. Thus, they are not applicable to bugs such as Tcas V25 (row 35), where the predicate is a boolean expression but not the conditional statement of an if statement. There are other types of bugs, such as Replace V23 (row 16) and Totinfo V5 (row 38), that might be fixed by SPR and Prophet using a schema called Copy and Replace, which copies an existing statement in the code before another statement and replaces a value in this statement with a different one. However, this schema can fix these two bugs only if it can find the right statements to be copied.

The repair operators of Doctor Code do not have any of the above limitations.

To compare Doctor Code with Prophet in terms of both the number of overfitting patches and repair time, although Prophets' benchmarks are different from our benchmarks, we used Prophet's generalized results for comparison with Doctor Code. Within the 12-hour time limit, Prophet found plausible patches for 39 of the 69 defects. The authors of [10] found correct patches for 18 of these 39 defects. In addition, they mentioned that for the 15 defects, the first validated patch is correct. Therefore, based on its benchmarks, about 40% of plausible patches were correct, and about 60% of them were overfitted. Also, the authors of [8] reported that Prophet required an average of 108.9 and 138.5 minutes to find and validate the first plausible patch and the first correct patch, respectively. As mentioned above, Doctor Code requires 90 minutes to find and validate all plausible patches, 90% of which are correct.

- *Comparison with APR tools for Java programs*

  Some of the famous tools that are evaluated using Defects4j are presented in Table 13. The columns show the number of correct and incorrect patches produced by each tool. The following machine-learned tools are analyzed in this table.

  The main difference between our method and other state-of-the-art machine learning tools (such as Elixir, Prophet, and GenProg) is the selected repair operators and their prioritization model. Due to these differences, Doctor Code significantly improves program repair compared to competing tools.

  We have evaluated Doctor Code's ability to generate correct patches against seven automated approaches on the Defects4j dataset. According to the results in Table 13, Doctor Code outperforms its competitors in terms of the number of fixed bugs: 37 (Doctor Code) versus 4 (HD-Repair), 18 (ACS), 5 (Nopol), 27 (SimFix), 5 (jGenProg), 26 (Elixir) and 23 (DLFix). Doctor Code also outperforms the other tools in terms of the ratio of overfitted patches to total patches: 18% (Doctor Code) versus 87% (HD-Repair), 21% (ACS), 85% (Nopol), 77% (SimFix), 81% (jGenProg), 36% (Elixir) and 57% (DLFix).

5. Conclusions

This paper presented Doctor Code, a new test suite-based Automated Program Repair (APR) technique that prioritizes repair operators by extracting features from buggy programs and using machine learning. Doctor Code operates in two phases: training and deployment. In the training phase, an Multinominal Logistic Regression (MLR) model is trained, which is then employed in the deployment phase. Using this model and feature vectors extracted from buggy programs, Doctor Code prioritizes its repair operators and produces candidate patches by applying them to buggy programs in the deployment phase.

Also, we have demonstrated that Doctor Code can produce patches that do not exist in the search space of Semifix, SPR, Prophet, Elixir, and DLFix. To investigate the impact of machine learning on prioritizing repair operators, we also implemented another APR technique called Random APR (RAPR), in which the repair operators of Doctor Code are applied to the buggy program in random order. Comparing RAPR and Doctor Code indicates that using machine learning to prioritize repair operators declines the repair time and the number of overfitted patches by 82:68% and 33:33%, respectively. Doctor Code utilizes an MLR model to prioritize repair operators. We investigated the impact of using other classification models on this task. The experimental results indicate that although MLR performed better than the other models in terms of running time, Doctor Code has almost the same performance for all four prediction models.

Since Doctor Code is effective at prioritizing repair operators, In future work, we plan to enrich its repair operator repository by adding several newly designed repair operators and even those utilized by other APR techniques. Also, we plan to design repair

operators that fix bugs occurring at multiple points within programs. Our future work also includes evaluating Doctor Code on real-world bugs and combining its repair operator prioritization capability with other APR techniques, such as [6,7] that employ repair operators in a predefined order.

Acknowledgments

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

1. Wang, Y., Yang, J., Lou, Y., et al., *Attention: Not Just Another Dataset for Patch-Correctness Checking*, arXiv preprint arXiv:2207.06590 (2022).
2. Tassey, G., *The Economic Impacts of Inadequate Infrastructure for Software Testing*, National Institute of Standards and Technology (2002).
3. Marshall, I.J. and Wallace, B.C. "Toward systematic review automation: A practical guide to using machine learning tools in research synthesis", *Systematic Reviews*, **8**, pp. 1–10 (2019).
4. Goues, C.L., Nguyen, T., Forrest, S., et al. "Genprog: A generic method for automatic software repair", *IEEE Transactions on Software Engineering*, **38**(1), pp. 54–72 (2012).
5. Nguyen, H.D.T., Qi, D., Roychoudhury, A., et al. "Semfix: Program repair via semantic analysis", in: *Proc. International Conference on Software Engineering*, pp. 772–781 (2013).
6. Goues, C.L., Pradel, M., and Roychoudhury, A. "Automated program repair", *Communications of the ACM*, **62**(12), pp. 56–65 (2019).
7. Long, F. and Rinard, M. "Staged program repair with condition synthesis", in: *Proc. 10th Joint Meeting on Foundations of Software Engineering*, pp. 166–178 (2015).
8. Long, F. and Rinard, M. "Automatic patch generation by learning correct code", in: *Proc. 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 298–312 (2016).
9. Perkins, J.H., Kim, S., Larsen, S., et al. "Automatically patching errors in deployed software", in: *Proc. ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pp. 87–102 (2009).
10. Carzaniga, A., Gorla, A., Mattavelli, A., et al. "Automatic recovery from runtime failures", in: *Proc. International Conference on Software Engineering*, pp. 782–791 (2013).
11. Wang, S., Wen, M., Lin, B., et al. "Automated patch correctness assessment: How far are we?", In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 968–980 (2022).
12. Qi, Z., Long, F., Achour, S., et al. "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems", In *Proceeding of International Symposium on Software Testing and Analysis*, pp. 24–36 (2015).
13. Le, X.B.D., Le, T.D.B., and Lo, D. "Should fixing these failures be delegated to automated program repair?", in: *Proc. IEEE 26th International Symposium on Software Reliability Engineering*, pp. 427–437 (2015).
14. Saha, R.K., Lyu, Y., Yoshida, H., et al. "Elixir: Effective object-oriented program repair", in: *Proc. 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 648–659 (2017).
15. Do, H., Elbaum, S., and Rothermel, G. "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact", *Empirical Software Engineering*, **10**(4), pp. 405–435 (2005).
16. Just, R., Jalali, D., and Ernst, M.D. "Defects4j: A database of existing faults to enable controlled testing studies for java programs", in: *Proc. International Symposium on Software Testing and Analysis*, pp. 437–440 (2014).
17. Arcuri, A. "On the automation of fixing software bugs", in: *Proc. Companion of the 30th International Conference on Software Engineering*, pp. 1003–1006 (2003).
18. Ghosh, D. and Singh, J. "Spectrum-based multi-fault localization using chaotic genetic algorithm", *Information and Software Technology*, **133**, p. 106512 (2021).
19. Weimer, W., Fry, Z.P., and Forrest, S. "Leveraging program equivalence for adaptive program repair: Models and first results", in: *Proc. 28th IEEE/ACM International Conference on Automated Software Engineering*, pp. 356–366 (2013).
20. Kim, D., Nam, J., Song, J., et al. "Automatic patch generation learned from human-written patches", in: *Proc. International Conference on Software Engineering*, pp. 802–811 (2013).
21. Monperrus, M. "A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair", in: *Proc. 36th International Conference on Software Engineering*, pp. 234–242 (2014).
22. Debroy, V. and Wong, W.E. "Using mutation to automatically suggest fixes for faulty programs", in: *Proc. 3rd International Conference on Software Testing, Verification and Validation*, pp. 65–74 (2010).
23. Jones, J.A. and Harrold, M.J. "Empirical evaluation of the tarantula automatic fault-localization technique", in: *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 273–282 (2005).

24. Abreu, R., Zoeteweij, P., Golsteijn, R., et al. "A practical evaluation of spectrum-based fault localization", *Journal of Systems and Software*, **82**(11), pp. 1780–1792 (2009).

25. Xuan, J., Martinez, M., DeMarco, F., et al. "Nopol: Automatic repair of conditional statement bugs in Java programs", *IEEE Transactions on Software Engineering*, **43**(1), pp. 34–55 (2017).

26. Li, Y., Wang, S., and Nguyen, T.N. "Dlfix: Context-based code transformation learning for automated program repair", In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 602–614 (2020).

27. Lee, A.H. and Silvapulle, M.J. "Ridge estimation in logistic regression", *Communications in Statistics-Simulation and Computation*, **17**(4), pp. 1231–1257 (1988).

28. Hall, M., Frank, E., Holmes, G., et al. "The Weka data mining software: An update", *SIGKDD Explorations Newsletter*, **11**(1), pp. 10–18 (2009).

29. Barr, E.T., Brun, Y., Devanbu, P., et al. "The plastic surgery hypothesis", in: *Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 306–317 (2014).

30. Campos, E.C. and Maia, M.d.A. "Common bug-fix patterns: A large-scale observational study", in: *Proc. ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 404–413 (2017).

31. Parsa, S., Vahidi-Asl, M., and Asadi-Aghbolaghi, M. "Hierarchy-debug: a scalable statistical technique for fault localization", *Software Quality Journal,* **22**(3), pp. 427–466 (2014).

32. Parsa, S., Mousavian, Z., and Vahidi-Asl, M. "Analyzing program dynamic graphs for software fault localization", in: *Proc. 5th International Symposium on Telecommunications*, pp. 169–174 (2010).

33. Sokolova, M. and Lapalme, G. "A systematic analysis of performance measures for classification tasks", *Information Processing and Management*, **45**(4), pp. 427–437 (2009).

Biographies

Sharmin Moosavi is a PhD student in the Faculty of Computer Science and Engineering at Shahid Beheshti University, Tehran, Iran. She received her BSc and MSc from the Software Engineering Group of the Department of Computer Engineering, Isfahan University, Isfahan, Iran. In addition, she is a faculty member of Islamic Azad University. Her research interests include serious games, computation games, software testing, and debugging.

Mojtaba Vahidi Asl is an Assistant Professor in the Computer Science and Engineering faculty at Shahid Beheshti University. He received his BSc from the Amirkabir University of Technology in 2005 and his MSc and PhD from the Iran University of Science and Technology in 2008 and 2014, respectively. His research interests include Software testing, Fault localization, Program repair, and Computer Games.

Hassan Haghighi received his PhD degree in Computer Engineering-Software from Sharif University of Technology in 2009 and is currently a Professor in the Faculty of Computer Science and Engineering at Shahid Beheshti University, Tehran, Iran. His main research interests include formal methods, software testing, and data quality.

Mohammad Rezaalipour is a PhD student in the Software Institute, part of the faculty of Informatics of the Università della Svizzera Italiana (USI). He develops tools and techniques for the debugging of data science programs.